Problem 6. A Sudoku solver using Genetic Algorithms

Victor Montiel Argaiz

May 31, 2012

Contents

1 Introduction **3**

2 Description of the genetic algorithm **4**
2.1 Genotype Encoding 4
2.2 Fitness Evaluation 4
2.3 Initialization 5
2.4 Diversity Measure 5
2.5 Mutation 6
2.5.1 Swap Mutation 6
2.5.2 3-Swap Mutation 6
2.5.3 Intelligent Mutation 6
2.5.4 Comparative performance of mutator operators 9
2.5.5 Combined Mutation 10
2.6 Conclusion of mutation methods 10
2.7 Crossover 12
2.7.1 Box Crossover 12
2.7.2 Optimized Row-Column Crossover 14
2.7.3 Combined Crossover 16
2.7.4 Conclusion of Crossover methods 16
2.8 Selection 18
2.8.1 Elitist methods 18
2.8.2 Comparative Results 18
2.9 Termination condition 19
2.10 Local Search 19

3 Code Structure and Implementation **21**

4 Final Results **23**
4.1 Basic Solution, mutation and crossover 23
4.2 Local Search 24
4.3 Parameter adaptation 26
4.4 Conclusion 26
4.4.1 Improvements 27

5 Appendix **28**

A Program Compilation **28**

B Program Execution **28**

C Sudoku Puzzles **29**

D Acronyms **29**

1 Introduction

Sudoku is a mathematical puzzle consisting of a partially completed 9×9 grid, divided in 9 boxes, each defining a 3×3 grid. A valid solution consists of an arrangement of numbers from 1 to 9, so that no number is repeated on any of the rows, columns and boxes. On its initial state, the Sudoku board, as shown in figure (1) has a number of cells specified (the *givens* or *clues*), so that the puzzle is well-defined, i.e, has a unique solution.

	2			3		9		7
	1							
4		7				2		8
		5	2				9	
			1	8		7		
	4				3			
				6			7	1
	7							
9		3		2		6		5

Figure 1: An example of a Sudoku Puzzle

The problem, that can be expressed as the better-known *coloring problem*, is an example of an *exact cover problem*. The search of the solution of a well-defined puzzle can be approached with *combinatorics* and *permutation group theory* from a mathematical point of view, where a large number of algorithms in these areas have been used to solve these kind of problems. The general case of solving Sudoku puzzles on $n^2 \times n^2$ boards is known to be *NP-Complete*. For the usual $n = 3$ size, the one we are tackling in this exercise, this is not bad news, since algorithms from the family of *Constraint Satisfaction Problems (CSP)*, such as *Dancing Links* [3], *Backtracking* and *Constraint Propagation* can find a solution in the blink of a second. However, for larger sizes, the problem can be tough. For the classic 9×9 board, the number of possible solutions has been shown to be 6.670.903.752.021.072.936.960, so, for larger sizes, even sophisticated algorithms fail to give a solution in a reasonable amount of time.

In this project we are aimed to find a solution for Sudoku Puzzles using techniques from *evolutionary computing* family, specifically *Genetic Algorithm (GA)*, even if these techniques are not the best suited for CSP. In order to test the robustness of the algorithm, we have selected five Sudoku puzzles for each level: Easy, Medium, Hard and Very Hard from the `www.dailysudoku.com` website. We have used Easy and Medium levels to understand the behavior of the different configurations of the GA, and once well-suited parameters have been found, we have used these parameters to solve the most challenging puzzles.

2 Description of the genetic algorithm

2.1 Genotype Encoding

Individuals in our solution are represented in a linear array of 81 elements, each having a number from 1 to 9. Numbers are arranged in boxes as in figure (2), so that elements within the same box are in consecutive positions in memory. Since many operators act on a box domain, this arrangement could eventually improve the performance of memory operations, showing some speed-up in the execution versus the other usual bi-dimensional array representation matching the visual order.

Each box contains a valid permutation of the numbers from 1 to 9, being a valid permutation one where no number is repeated and *givens* values are at their initial positions. Observe that by forcing this condition by the definition of the genotype, the *box constraint* is never violated. Given that GA are not the best method to solve CSP, the imposition of this restriction at box-level by the definition of the genotype will eventually help guiding faster the evolutionary process to the solution, since one of the constraints is always satisfied.

1	2	3						
4	5	6		2			3	
7	8	9						
	4			5			6	
	7			8			9	

1_1	2_1	3_1	4_1	5_1	6_1	7_1	8_1	9_1	1_2	$\cdots$	1_3	$\cdots$	4_9	5_9	6_9	7_9	8_9	9_9

Figure 2: Box numbering and memory representation as a linear array. The subscript number represents the box index, while the number represent the index of the element within the box, following the convention shown in the Sudoku board above

2.2 Fitness Evaluation

As mentioned on the introduction, a Sudoku puzzle is an example of a *CSP* where the *feasibility condition* is made up of three *constraints*: the restriction of uniqueness of numbers at box-level, at row-level and at column-level. A solution is meant to be valid only if it meets the *feasibility condition*, i.e, fulfills the three restrictions. If we approach the Sudoku problem as a pure *CSP*, the evaluation of each individual will take only two values, *true* or *false*, whether the *feasibility condition* is met or not. However, such

an evaluation function would be helpless for a *GA* since the landscape of the function over the search space would be a flat plateau with *false* value and one unique point, the solution, with value *true*. Such a landscape with no clues for the *GA* would make hard the optimization process. Typically, as explained in [2], when dealing with *CSP* in *GA*, we usually approach the problem both with a fitness function which counts the number of constraints violated in the explored solution and with an encoding of the genotype which enforces all or some of the constraints.

In our solution, the *box constraint* is met by the definition of the genotype, as we have seen on the section above, so we do not need to include this constraint on the evaluation function. Thus, only the other two constraints for rows and columns are left to be considered by the *fitness function*.

A straightforward *fitness function* consists then in summing the number of violated constraints. Thus, for each missing or repeated number in each row or column, we add one to the function. A mathematical definition of this *fitness function* is found on equation 1, where x_{ij} is the cell at row i and column j, $x_{i\cdot}$ is row i and $x_{\cdot j}$ is column j, and the $Rep(A)$ function returns the number of duplicate elements within the set A.

$$f(x) = \sum_{i=1}^{9} Rep(x_{i\cdot}) + \sum_{j=1}^{9} Rep(x_{\cdot j}) \tag{1}$$

2.3 Initialization

The initialization of the individual at the beginning of the execution of algorithm is made randomly. There exist other approaches in which individuals are initialized according to some heuristics based on the initial *givens*. However, this initial individuals could eventually skew the search process towards a non-optimal solution. To assure we make a broad enough search, we have implemented the random method.

On a first stage, individuals are initialized with the *givens* of the problem. Then, each position box_i which is not a given value is filled with a random shuffle of the set $\{1,2,3,4,5,6,7,8,9\} - Givens_i$, where $Givens_i$ are the *givens* for the box_i. As discussed, by initializing the individuals satisfying the box constraint, and assuring that the transformation operators do not lead to individuals violating this constraint, the *GA* has to work only on the minimization of the two other constraints.

2.4 Diversity Measure

Diversity within the population of an Evolutionary Computing (EC) algorithm is crucial for the success of these techniques. A bigger diversity will explore a broader front of potential solutions and hopefully will lead to the optimal solution, avoiding getting stuck on local minima. In order to keep track of the diversity of the population, which will be used to closely monitor the evolution of the algorithm, we have developed a method which allows us to calculate the number of different individuals within the current generation.

The straightforward way to do it is to compare individuals by pairs, checking how many of them are different. However, this method can be time-consuming, specially for large populations. We have thus thought of an alternative hashing method to get an estimation of the diversity without consuming an important fraction of the cpu-time to just monitor the evolution of the algorithm.

For each individual, we have calculated a hash number. The hashing function is based on the *rotating hash* method [6], which is a simple way to get a minimal acceptable mixing. Each box is converted to a 9-digit number and then we get the hash key for each individual by shifting and mixing the 9-digit

numbers of each of the 9 boxes. Once we have all hashes for each individual, assuming the hashing function does not contain collisions, we can easily identify in $log(n)$ time the number of different individuals using containers such as the C++ `set`. Since the calculation of the hashing function is made up of basic arithmetic operations the current implementation allows us to measure the diversity of the population without the overhead in cpu-time which would be introduced if we compare instead each individual by pairs.

2.5 Mutation

In this problem we have considered three mutation methods that eventually can be combined in a global mutator, which randomly chooses the operator to apply at each individual. For the mutation operator we have designed both, completely *random operators*, the classical ones used for permutation-type genotypes, and *hybridization techniques* which employs intelligent variation operators. These operators, instead of behaving 100% randomly, use problem-specific knowledge in order to carefully choose or assign more probabilities to the gen to be mutated according to some heuristic adapted to the problem to be solved.

It is important to mention that all mutation operators are applied at a box-level so that after the mutation, the *box constraint* that we have enforced is not violated. Furthermore, *givens* cells cannot be transformed by the operator, since they are part of the definition of the problem.

2.5.1 Swap Mutation

Swap mutation is the traditional random operator used in most permutation-based genotypes, as explained in [2]. It randomly chooses two positions within a box and swap their content, as long as none of the positions is a given value. By construction, the use of this operator cannot break box-level constraints. However, its application might cause violation of constraints at the row and column level. The worst case is when two genes (positions in the grid) satisfy the row and column constraints at their initial position and after the application of the swap mutation they violate these constraints, worsening the fitness value of the individual by four points.

Thus, intuitively, the application of this operator could be more useful at the first stages of the execution where we are most interested in doing a broad search than at the final stages where we are close to a solution and its application might cause up to four constraints violated.

2.5.2 3-Swap Mutation

The 3-swap mutation is similar to the one above, with the only difference that instead of swapping two genes, we shift three genes. Again, box-level constraint is satisfied by construction of the operator, since we only shift elements within the box, which are not repeated, and *givens* are preserved at their original positions.

2.5.3 Intelligent Mutation

This operator implements one of the many *hybridization techniques* that can be applied to *GA*. By using specific knowledge of the problem to be solved, we can create mutation operators which bias the path of the evolution to a region of the search-space where we think the optimal solution might be, according to some heuristic. This use of specific knowledge of the problem gives the name to *intelligent operators*. Observe that *intelligent operators* do not need to be always deterministic, instead, some random behavior can be conserved to avoid biasing the search to a local optimal solution.

In the case of the Sudoku Puzzle, we have designed an intelligent operator that look for constraint violations, either at row or column level, choose one violation randomly, and try to fix the violation by

swapping specific gens within the same box. This algorithm uses a *greedy* approach as the heuristic, since it fixed the first violation it finds even if it is not the optimal movement.

To implement this operator we have first to look for the constraint violations. To this purpose, we scan the grid saving the missing and duplicated elements at each row and column. Once we have detected all violations, we choose randomly one of the rows or columns where the violation appears, and look for the duplicated element. After identifying the box where the duplicated element is, we look within the same box for the missing element of the column, and swap the two positions. Observe that, before applying this operator, we have to check that the duplicated and missing element to be swapped are not a *given* element within the box. Also, notice that, by construction, since the swap of elements is made within the same box, the box constraint is always conserved.

The application of this operator allows to guide the mutation operator to create a new individual with more constraints satisfied. Observe, however, that the since we are just fixing either row or column constraint violations, the mutation that fix the row violation might have, at the same time, the potential to introduce a new column constraint violation. Thus, not all executions of this operator must lead to an individual with better fitness value. Furthermore, even if the application of the operator does not introduce a violation of constraints within the row/column where it was applied, it has the potential to introduce it in other distant rows/columns of the grid. Hence, as it usually happens within the *GA* methods, the specificity of the operator might introduce a bias to a locally optimal board, instead of guiding the algorithm toward the solution. This trade-off between breadth of search and quickness to find the solution has always a delicate equilibrium point. An example of execution of this operator can be found on figure (3).

2	5	4	3	7	9	8	1	9
3	1	6	8	9	5	2	4	7
8	9	7	2	4	1	3	5	6
6	2	8	9	1	7	5	3	2
9	3	5	4	6	8	1	7	4
4	7	1	5	3	2	9	6	8
1	4	3	7	2	9	6	8	5
7	8	2	6	5	3	4	9	1
5	6	9	1	8	4	7	2	3

2	5	4	3	7	9	8	1	9
3	1	6	8	9	5	2	4	7
8	9	7	2	4	1	3	5	6
6	4	8	9	1	7	5	3	2
9	3	5	4	6	8	1	7	4
2	7	1	5	3	2	9	6	8
1	4	3	7	2	9	6	8	5
7	8	2	6	5	3	4	9	1
5	6	9	1	8	4	7	2	3

Figure 3: Example of the intelligent mutation operator. At row 4, the element 4 is missing, and element 2 is duplicated. At row 5, element 2 is missing and element 4 is duplicated. We choose randomly the violated constraint at row 4, then we detect that element 2 is duplicated and 4 is missing. We swap the elements 2 and 4 within the same box, assuming none of them is a given cell. Observe that this transformation has solved the constraint violation at row 4, but has introduced new violations at columns one and two.

2.5.4 Comparative performance of mutator operators

Once the three implemented mutation operators have been introduced, we have to evaluate their relative performance in order to carefully choose their probabilities of execution within the combined mutation operator which will include the three above methods.

To measure the performance of each operator, we have chosen an *Easy* and *Medium* level puzzles. Execution configuration files can be found on `mutation-test-XX.cfg`, `mutation-test-medium-XX.cfg`. The experiment tries to solve the `Easy-1.txt, Medium-1.txt` puzzles by just applying mutation. With a fixed mutation probability, we apply just one of the operators at each experiment. In a final experiment within the configuration file `mutation-test-04.cfg, mutation-test-medium-04.cfg`, we combine the three operators with relative probabilities according to their performances. To measure the performance, we take into account the Success Rate (SR), Mean Best Fitness (MBF) and Mean Number of Fitness Evaluation To Solution (MNFES). In theory, the absolute performance for this particular problem should be measured against the Success Rate (SR) statistics, since the result can be either a valid Sudoku solution or an invalid one. However, we have also kept track of the MBF statistic to measure, for those cases where the solution was not found, how close to the solution we stayed, according to the fitness function. Finally, MNFES statistic will allow us to measure the time needed to reach a solution for a given configuration.

Results are shown on table (1). As we can see, for a small population of $N = 300$ individuals and after a maximum of 2000 generations, the mutator operators are able, by themselves, to solve an Easy-level puzzle. Indeed, in all of the 30 executions the mutation was able to find the correct solution, having all experiments $SR = 1.0$ and $MBF = 0.0$. Even if all three operators found the solution, the MNFES statistic shows a big difference in performance between them. As we had intuitively argued above, the Swap mutation is the worse performer, since, although it might be useful to do a breadth search at the first generations, once we have puzzles with just a few constraint violated, the application of this operator might break a lot of constraints. 3-Swap mutation shows a similar behavior, with just a little improvement in the results, needing less generations to find the solution. Finally, our intelligent mutation seems to be the best performer, reducing the number of individuals explored to achieve the solution by more than ten times. For the Medium-level puzzles, results are similar. Even if SR are near zero, that is, mutation operator is not able to solve by themselves medium-level puzzles, MBF values are lower for the intelligent mutation, meaning that the operator is able to find on average Sudoku boards potentially closer to the final solution than the other ones. Finally, the number of fitness evaluation needed to achieve the solution is also lower for this last operator, showing again its superiority.

Finally it is important to notice that for the Medium-Level puzzle, the results for the combined operator in terms of MBF and MNFES are better than just for the *intelligent* method, i.e., there exists some synergy between the different methods. This reinforce once again the idea that, although the *swap* and *3-swap* methods are less effective, their use on the first generations could improve the breadth of the search, being able to find best solution at the end of the process because of this initial larger exploration.

	Easy-Level			Medium-Level		
Operator	SR	MBF	$MNFES$	SR	MBF	$MNFES$
Swap	1.0	0.0	76630	0.067	9.733	290250
3-Swap	1.0	0.0	59130	0.067	9.733	108450
Intelligent	1.0	0.0	4850	0.033	3.767	89700
Combined	1.0	0.0	5980	0.067	3.067	74550

Table 1: Relative performance between mutation operators

2.5.5 Combined Mutation

Once we have obtained the above partial results for each operator we can then create a combined mutator assigning relative probabilities to each of the mutation methods. For subsequent experiments, we have used the following probabilities: $p_{swap} = 0.15$, $p_{3-swap} = 0.20$ and $p_{intelligent} = 0.65$. Even if intelligent mutation was by far the best performer within the previous experiment, we have considered interesting to maintain the other two operators with some probability, since, as we have seen in previous results, they can help with a breadth-search exploration at the beginning. In consequence, we have assigned the relative probabilities of execution of each of the operators according to their relative performance.

After deciding the configuration of the final *Combined Mutation* operator, we have to explore the behavior of the algorithm with different mutation probabilities and with puzzles of different difficulty. Experiments are made just considering the mutation operator, no crossover is performed in these simulations. Configuration files for experiments can be found on `mutation-test-mut-prob-XX.cfg` for the Easy-level puzzle and `mutation-test-mut-prob-med-XX.cfg` for the Medium-level puzzle. Figure (4) shows the results in terms of SR, MBF and MNFES for different mutation probabilities. For the Easy-level puzzle, we always get the solution, i.e., SR=1.0 and MBF is always null. In the case of the Medium-level puzzles, we can see a clear improvement of the results when probability of mutation increases, both in terms of SR and MBF. As for the MNFES, for the Easy-level puzzles results also improve with higher mutation probabilities, i.e., less generations are needed to achieve the solution, while for the Medium-level is the opposite, as the mutation probability increases, the generations needed to achieve the solution increases.

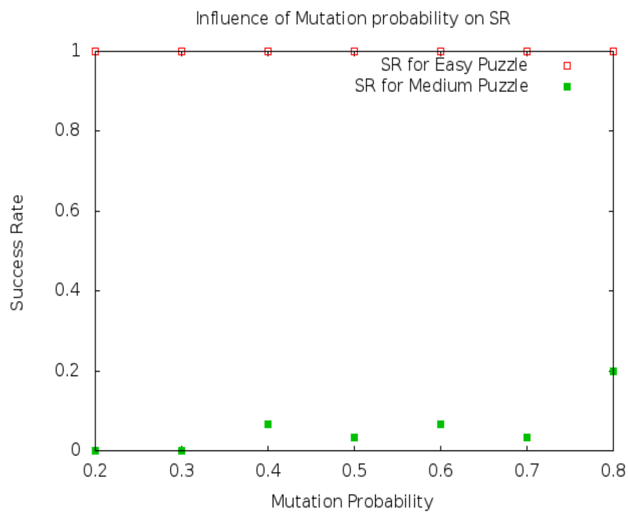
After a more detailed observation of the results of experiments, by exploring the log file for each execution, we see that the increase of the mutation probability in these experiments carries along an increase of the diversity of the population. This means that an increasing mutation rate makes a broader search, exploring more possible Sudokus, being able to identify the exact solution, avoiding being trapped in a local minimum, but requiring, on the opposite side, more time for convergence.

2.6 Conclusion of mutation methods

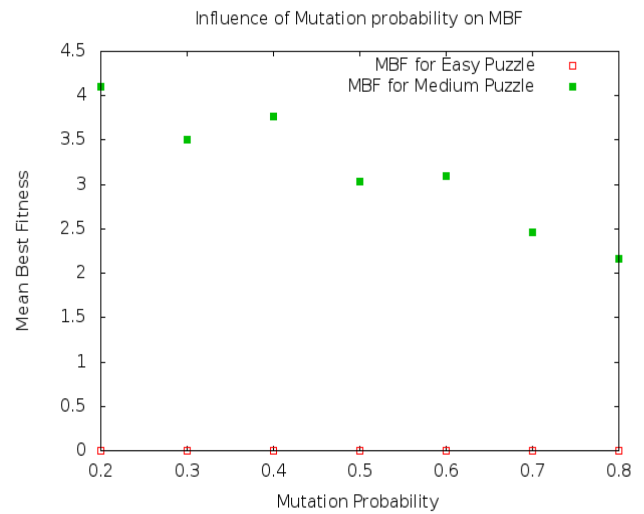
Summarizing the results on the three methods, the combined mutation method seems to behave better for the Sudoku Puzzle, by doing a broader search thanks to the *swap* and *3-swap* operators, and by specializing the search with an *hybridization technique* implemented for the *intelligent mutation*. This is an example of synergy amongst the operator, since by combining the three of them we get better results than taken each of them individually.

The GA can solve Easy-Level puzzles just by using mutation operator, i.e., without a crossover transformation. However, for Medium-Level (and superior), the mutation operator fails in most cases to find the final solution by itself. Figure (4) shows a weak dependency between the increase in mutation probability and SR and MBF, favoring simulations with higher mutation probability.

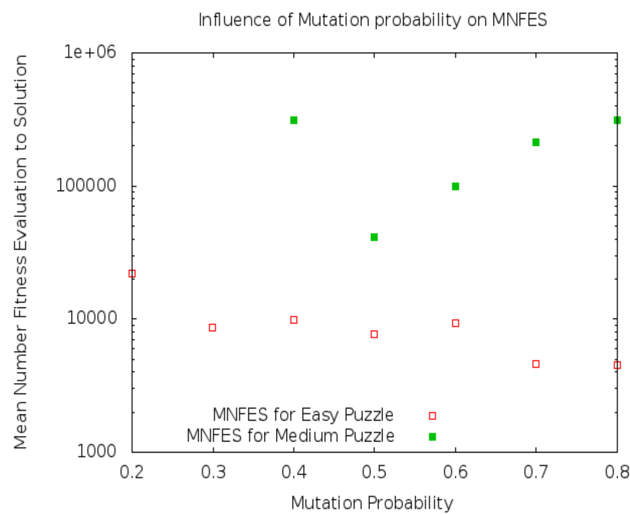
Finally, the increase in mutation probability carries along an increase of the diversity of the population, doing greater the chances of finding the solution. We have thus to work in parallel with the selection method to find an optimal equilibrium between the mutation probability and the diversity of the population we want to maintain to guide the algorithm.



(a) SR for different Mutation Probabilities



(b) MBF for different Mutation Probabilities



(c) MNFES for different Mutation Probabilities

Figure 4: Influence of mutation probabilities on Success Rate, Mean Best Fitness and Mean Number of Fitness Evaluation to Solution

2.7 Crossover

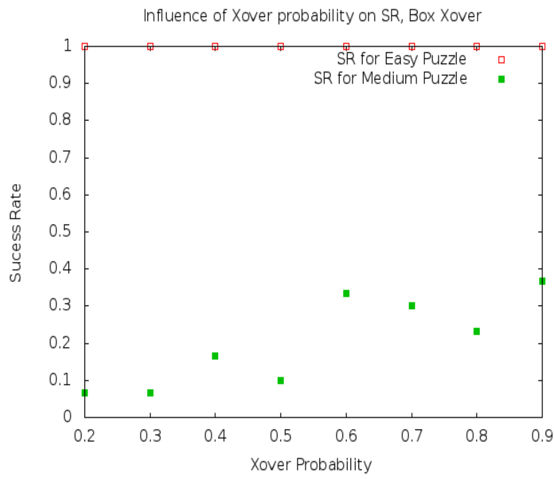
For the crossover operator, we have implemented, as for mutation, two different methods, one of them is completely random, and the other one fits in the *hybridization techniques.* In both techniques, we always do the transformation at a box-level, so that the box-constraint is respected after the application of the operator. Finally, a combined crossover operator has been created to make possible the execution of both operators within the same simulation according to a probability distribution specified by the user.

In order to draw conclusions of which method performs better, we have run similar experiments to those used with mutation operators. The experiments consist of 30 executions, each of one has a population of 300 individuals which evolve over 2000 generations.

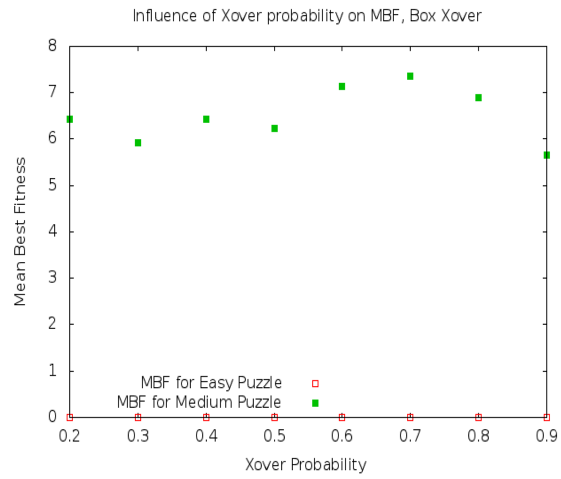
2.7.1 Box Crossover

The *box crossover* operator randomly swap the 3×3 boxes between the parents, creating two off-springs with boxes from each of the parents. Instead of implementing the usual *two-points* or *three-points* crossover in which we select a random point within the genotype and swap the genes to the left and right of that point, we have created a *multi-point* crossover method, in which, at each box, we select, according to a *box selection probability* whether the current box will be inherited from parent one or two.

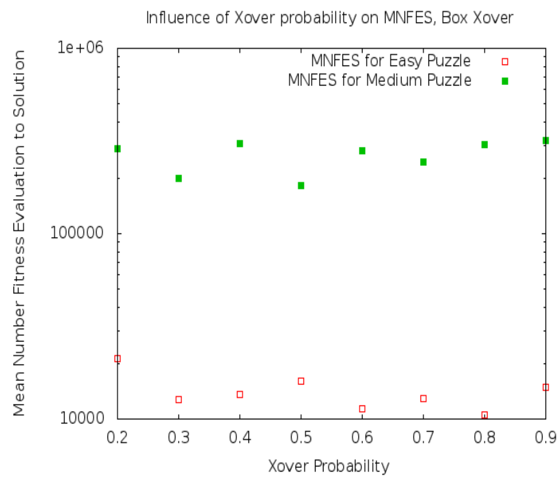
The performance of this operator can be seen in figure (5). As we can appreciate, the SR statistic seems to be higher for higher probabilities of crossover. The other two statistics, MBF and MNFES, do not exhibit any significant pattern with respect to the probability of crossover.



(a) SR for different CrossOver Probabilities



(b) MBF for different CrossOver Probabilities



(c) MNFES for different CrossOver Probabilities

Figure 5: Influence of CrossOver probabilities (box operator) on Success Rate, Mean Best Fitness and Mean Number of Fitness Evaluation to Solution

2.7.2 Optimized Row-Column Crossover

The *optimized row-column crossover* operator uses the knowledge of the problem to carefully choose the boxes we include on each of the off-springs. The method was introduced by [5]. After realizing that the usual swap crossover methods do not fit well within the Sudoku problem since the number of constraint broken after each crossover exceed by far the number of legal positions introduced, we came up with an operator which considers the constraints violated, both row and column-wise, at each of the parents.

The operator calculates the number of constraints violated at each row and column of the two parents. Then, it sums, for each parent, these numbers of violations at box level, ending up with 3 fitness values per parent which score each row of boxes, and 3 fitness values per parent which score each column of boxes. Once we have scored each row and column at box level, each offspring will be created according to the best score, either at column level or at row level (lower scores are best). For example, as shown in figure (6) offspring one will be created by choosing, column by column, the best columns-blocks of each of the parents, conserving their position in the original genotype. Offspring two will be created in the same way, but choosing row by row, the best row-blocks of each of the parents.

After the application of this non-randomized crossover operator, hopefully we will get off-springs with less constraints violated, either at column-level or at row-level. We expect thus that the performance of this method will be better with respect to the previous randomized method.

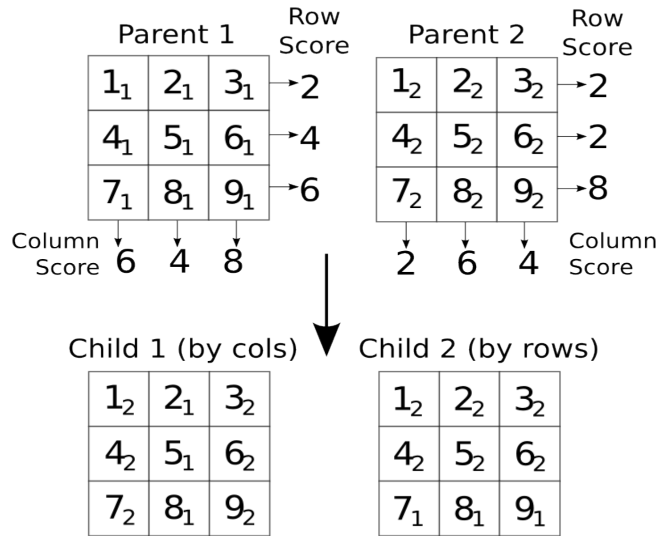
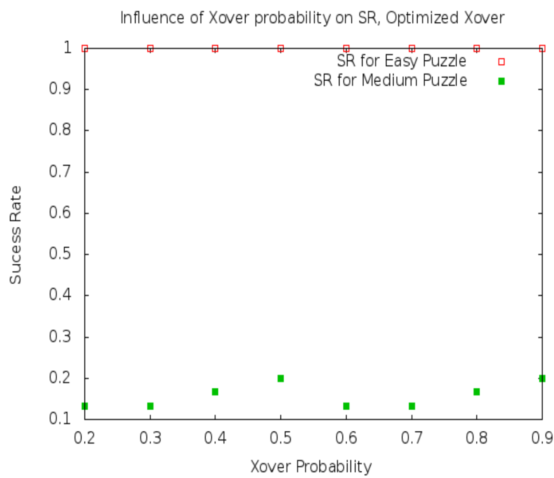
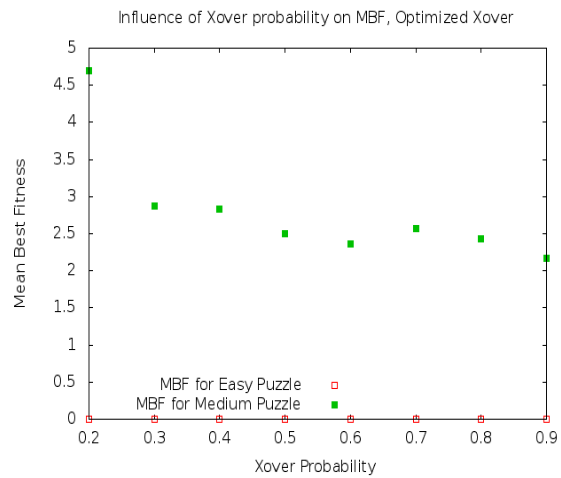


Figure 6: Hybridization techniques for Crossover operator

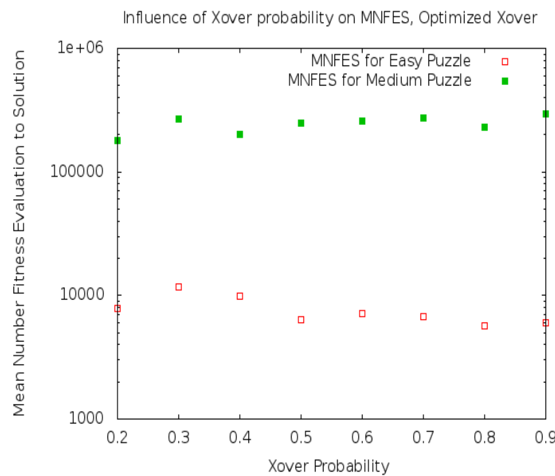
The performance of this operator can be seen in figure (6). In the three figures, I would not adventure to say that one probability behaves better than other. We see some peaks for the SR statistic on probabilities around $p = 0.5$ and $p = 0.9$, as well as a decrease in the MBF statistic around the same points. However, more simulations should be run in order to conclude with a stronger argument the dependency on the crossover probability for this operator.



(a) SR for different CrossOver Probabilities



(b) MBF for different CrossOver Probabilities



(c) MNFES for different CrossOver Probabilities

Figure 7: Influence of CrossOver probabilities (optimized operator) on Success Rate, Mean Best Fitness and Mean Number of Fitness Evaluation to Solution

2.7.3 Combined Crossover

As we did for the mutation operator, we have built a Crossover operator which combines the two above mentioned methods, according to a given probability distribution. This way, we can have the best properties of each of the method. The *combined operator* will be executed according to the crossover probability, and, whenever the crossover is performed, either one operator (*swap*) or another (*optimized*) will be applied according to their relative probabilities.

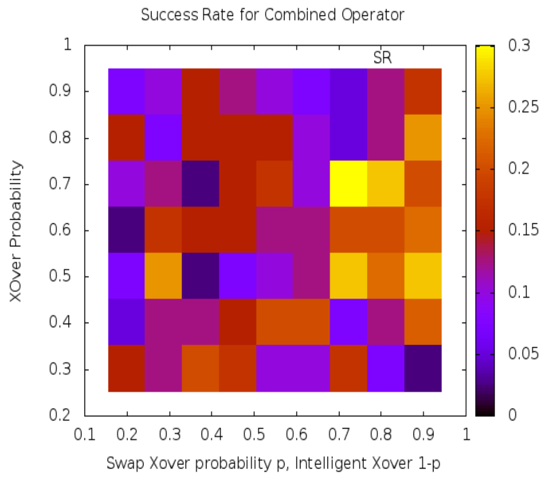
Observe that to fine tune the *combined operator* we have, not only to select a crossover probability, but also the individuals probabilities of each operator. In order to have an insight of which configuration is better, we have run several tests varying the probability of crossover and the probability of each operator. The test run a set of 40 executions of up to 3000 generations within a Hard-Level Sudoku. Configuration files for this test can be found on `xover-combined-hard-prob-XX-XX.cfg`, and results can be seen in figure (8).

For the SR statistic, we can identify a region at the middle right of the color map with yellow tones with SR up to 30%. For the MBF statistic we can identify diagonal bands of blue tones with minimal values for this parameter. Finally, for the MNFES we can not really appreciate any significant pattern. Following these results, we identify the parameters $p_{xover} = 0.7$, $p_{swap-xover} = 0.7$, $p_{intelligent-xover} = 0.3$ as one of the most suitable for this problem.

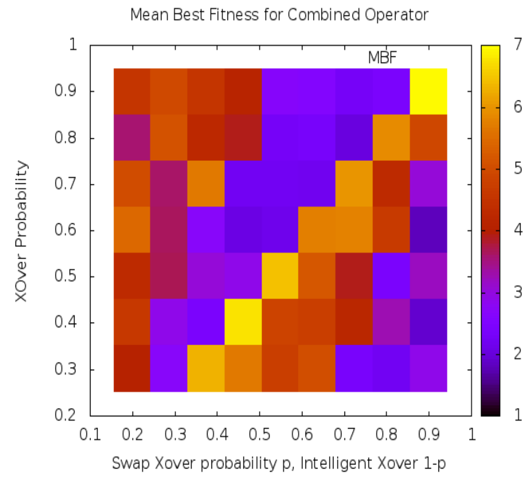
2.7.4 Conclusion of Crossover methods

Figure (9) shows the relative performance for the two implemented crossover method. Surprisingly, in terms of SR, the performance of the simple *swap crossover* is better than that of the *optimized row-column crossover*, specially for large parameters of $p_{crossover}$. However, we obtain better MBF with the more advanced operator. The conclusion to be drawn is that, while the *optimized row-column crossover* method improves the average fitness of the population, consequently getting better individuals, it seems to get stuck often in local-minima since the SR statistic is below the one obtained with the random *swap crossover*.

The final combined operator which will be used in the final configuration have a near-optimal mixing of both method which considerably improves the performance of each of them taken apart, getting SR of up to 0.3 for Hard-Level problems, as we can appreciate in the color map at figure (8). This implies a small increase on the performance of the GA with respect to the exclusive use of *mutation* as variational operators, where we hardly were able to solve Medium-Level problems.



(a) SR for different CrossOver Probabilities



(b) MBF for different CrossOver Probabilities

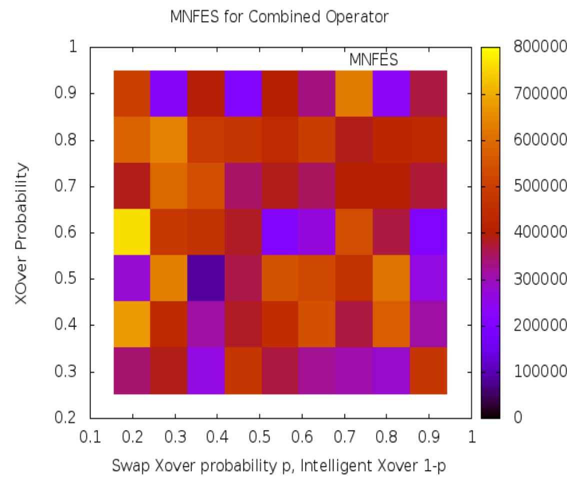
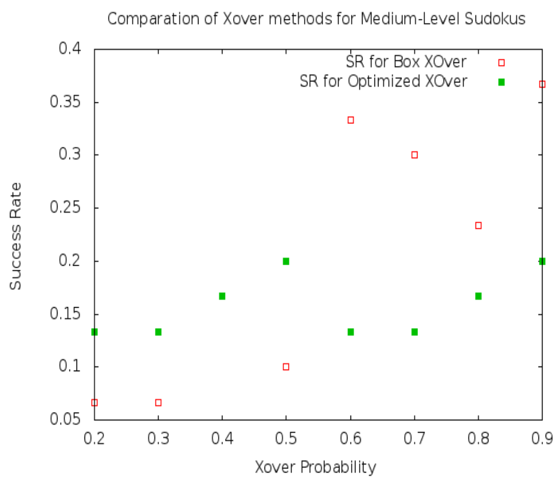
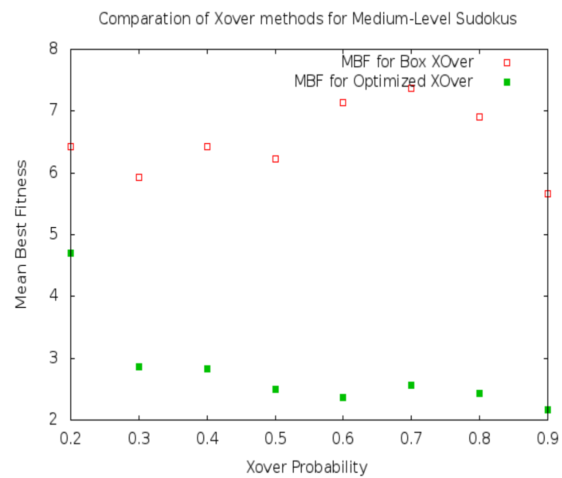


Figure 8: Influence of CrossOver probabilities on Success Rate, Mean Best Fitness and Mean Number of Fitness Evaluation to Solution



(a) Comparative of SR for the two crossover methods



(b) Comparative of MBF for the two crossover methods

Figure 9: Comparative of the two crossover method for Medium-Level Sudokus

2.8 Selection

Selection method has the responsibility of choosing the individuals which will mate to create the offsprings and of picking the individuals which will survive for the next generation. The selection method must provide a minimum degree of diversity to the population, to assure that we explore as much as individuals as possible, and, at the same time, retain at each generation the best individuals found until then.

In order to study which is the selection method which perform better for the Sudoku problem, we have implemented the *fitness proportional* and *ranking selection* as the simplest algorithms and the *stochastic tournament with elitism* and *ranking selection with elitism* methods to see if elitism introduce a significant performance. As in previous experiments, given that the Easy-Level puzzles seems to be easily solved by the basic operator, we have focused on the performance of the selection method for Medium and Hard-Level puzzles.

Fitness proportional and *Ranking selection* have been previously implemented in the previous problem sets, so, a detailed description of these method will not be provided here. Remember that a *shape-factor* was introduced in both methods, with respect to the basic implementation, so that we can adjust the selection pressure of each operator.

2.8.1 Elitist methods

The *Stochastic Tournament with Elitism* and *ranking selection with Elitism* have been developed with the aim of trying to improve the results obtained with the two previous methods by maintaining in the simulations the eternal balance between exploration and conservation of best individuals.

Elitist selection methods always retain the best individuals within the population, typically a proportion α of the entire population, while leaving the other $1 - \alpha$ behaving as an ordinary selection method. By retaining the best individuals we assure that fitness value are always improving or constant, with no chance of *involution* or *regression*. At the same time, by letting the remainder of individuals follow the ordinary selection pressure, we will enable the algorithm to explore new individuals.

The method has been implemented using both *Stochastic Tournament* and *Ranking* selection algorithms as described in [2], in which we can set up the size of the tournament and the probability of choosing the best individual at each tournament. With these two parameters we are able to adjust the selection pressure on the $1 - \alpha$ proportion of individuals not retained by the elitist part of the algorithm.

2.8.2 Comparative Results

Results with the four methods have not been so successful as we expected. For *Fitness proportional* method, we failed to find an optimal *shape-factor* behaving considerably well for Easy-Level puzzles. The selection method was unable to retain the best individuals within the population, achieving in all cases a high degree of diversity but poor average and minimum fitness values, obtaining SR values near to 0.0 even for the most simple puzzles.

Ranking selection method has been proven to be the best algorithm of the four for the Sudoku problem. While the selection method behaves very differently according to the *shape-factor* parameter, we managed to find a value for which we get pretty good results for the Easy-Level (very close to SR of 1.0), and for the Medium-Level puzzles. Outside this *shape-factor* value, the method failed to cast good results numbers.

Finally, the two implemented *elitist* methods also failed to achieve good results even for the Easy-Level puzzle. Test were made with α values from 0.05 to 0.25 and we did not appreciate any improvement with respect to the simpler *ranking selection* method. Since elitist method are more expensive in CPU-time, and do not improve significantly the results with respect to the *ranking selection* method, we have preferred to use this last for the final solution of the problem.

We think that the key factor behind the performance of the selection method is the diversity of the population. The Sudoku problem, as many other constraint satisfaction problems, has the particularity that we can eventually have individuals with low fitness values, but still being far from the final solution. After analyzing the log of the executions we have found many cases in which the algorithm get stuck with individuals with fitness values as low as 2, i.e., just two violated constraints, but which would eventually need many swaps between grids in order to convert them to the final solution. In these cases, a high diversity within the population, which allows the algorithm to explore many other solution niches, is a cornerstone for success.

2.9 Termination condition

Two termination conditions have been implemented for the GA. Firstly, the usual termination condition of the *maximum number of generation* achieved was developed. Under this termination condition, the algorithm stops evaluating new generations once the maximum number of generations has been achieved.

In order to avoid wasting time when a solution is found, we have implemented a second termination condition, *convergence condition* which finishes the simulation whenever a solution has been found. Since solution in the Sudoku problem is unique, once the solution is achieved there is no advantage to keep on running new generations to look for better solutions.

2.10 Local Search

As we have seen, GA is not the best method to solve Sudoku problems. Even if the Easy-level puzzles are solved quickly in most cases, Medium and Hard-level puzzles cannot be solved, in most cases, by traditional GA techniques. In order to improve the performance for difficult puzzles, we have decided to implement more specialized, and usually non-random, techniques such as *hybridization within variation operators*, explained in previous sections, and *local search.*

The idea of the introduction of a *local search* within a GA is simple, and similar to *local search* methods used at each step in standard optimization techniques. At each generation, once the *transformation operators* have been called, a *local search* method is invoked, bringing the individual to some other close individual with a better fitness value.

In our implementation, the *local search* make use of an analysis of the constraints of the Sudoku puzzles given the initial *hints* on the board. The idea consists mainly of enumerating the possible numbers that can be assigned to each cell on the grid, and is based on a very simplified version of the algorithms used to solve CSP. The data structure used to be used by the algorithm is similar to a Sudoku board, but instead, at each cell, we can store a set of possible valid numbers instead of a single guessed number. This data structure will be referred as *Constrained Sudoku Board.*

The first step is thus to build the *constrained sudoku board* starting with the *givens* as in figure (10). At each *given* position, the set associated to these cells will contain just the *given* number. Every other cell in the board will be initially filled with all numbers from 1 to 9.

Once we have set-up the initial *constrained sudoku board*, we propagate the constraints, reducing at each step the number of valid number at each cell. To do that, we traverse the grid cell by cell. At a

0	0	0	0	1	0	0	4	0
8	0	0	0	0	0	0	0	3
0	0	7	0	0	2	0	8	1
4	0	1	2	9	0	0	3	0
0	0	0	6	0	1	0	0	0
0	5	0	0	7	3	1	0	4
3	7	0	1	0	0	9	0	0
9	0	0	0	0	0	0	0	5
0	6	0	0	2	0	0	0	0

Figure 10: Initial puzzle with givens

given cell, we explore all cells in the same row, column and box as the cell we are studying. Then, for each cell in the column, row or box, we look for the sets that have a single element, i.e., cells that we know that the only possible number is the one on the set, and thus, it makes part of the final solution. All those numbers which are in single sets can be thus discarded from the set of possible values that the current cell can take, hence reducing the number of possible values that cell can hold.

This step can be repeated (propagated in CSP jargon) until the final *constrained sudoku board* does not change after a new step of propagation of constraints. Once we get this state, we will have a *constrained sudoku board*, as in table (2), in which there is a bunch of sets with single numbers, and other cells where the number of possible values they can take has been reduced considerably. What we have achieved with this method is thus to restrict the search of new individuals to the ones that satisfy the constraints specified in the *constrained sudoku board*. Since the search space is more limited, we expect that the number of individuals to be explored by the GA will be smaller, achieving the solution in a shorter period of time.

To end with the *local search* technique, we have to create a method such that, given an individual that has suffered the transformation operators, it returns a new individual which is the *projection* of the original one on the space of potentially valid solution described on the *constrained sudoku board* we have calculated. Here, *projection* has a similar sense as the *projection transformation* in linear algebra. The method looks for the closest individual to the original one that satisfies the *constrained sudoku board*. To achieve this projection we scan cell by cell the contents of each box, and we swap the elements within the same box until we achieve a box satisfying the box described in the *constrained sudoku board*.

Finally, because of its inner working method, we have called this *local search* as the *Projection to Derived Constraints* method.

Overall, the use of *local search* within the problem has improved considerably the performance of the algorithm. While we were not able to get SR statistics better than 0.3 for Medium and Hard-Level

$\{2,5,6\}$	$\{2,3,9\}$	$\{2,3,5,6,9\}$	$\{3,5,7,9\}$	$\{1\}$	$\{8\}$	$\{2,5,6,7\}$	$\{4\}$	$\{2,6,7,9\}$
$\{8\}$	$\{1\}$	$\{2,4,5,6,9\}$	$\{4,5,7,9\}$	$\{5,6\}$	$\{4,6,7,9\}$	$\{2,5,6,7\}$	$\{2,6,9\}$	$\{3\}$
$\{5,6\}$	$\{3,4,9\}$	$\{7\}$	$\{3,4,5,9\}$	$\{3,5,6\}$	$\{2\}$	$\{5,6\}$	$\{8\}$	$\{1\}$
$\{4\}$	$\{8\}$	$\{1\}$	$\{2\}$	$\{9\}$	$\{5\}$	$\{6,7\}$	$\{3\}$	$\{6,7\}$
$\{7\}$	$\{2,3,9\}$	$\{2,3,9\}$	$\{6\}$	$\{4\}$	$\{1\}$	$\{8\}$	$\{5\}$	$\{2,9\}$
$\{2,6\}$	$\{5\}$	$\{2,6,9\}$	$\{8\}$	$\{7\}$	$\{3\}$	$\{1\}$	$\{2,6,9\}$	$\{4\}$
$\{3\}$	$\{7\}$	$\{2,4,5,8\}$	$\{1\}$	$\{5,6,8\}$	$\{4,6\}$	$\{9\}$	$\{2,6\}$	$\{2,6\}$
$\{9\}$	$\{2,4\}$	$\{2,4,8\}$	$\{3,4,7\}$	$\{3,6,8\}$	$\{4,6,7\}$	$\{2,3,4,6\}$	$\{1\}$	$\{5\}$
$\{1\}$	$\{6\}$	$\{4,5\}$	$\{3,4,5,9\}$	$\{2\}$	$\{4,9\}$	$\{3,4\}$	$\{7\}$	$\{8\}$

Table 2: Constrained Sudoku Board deduced from the initial givens in figure(10)

problems with the traditional techniques, we manage now to achieve SR near to 1.0 for those levels, and reducing significantly the number of generations to the final solution. More detailed results and comparatives will be discussed within the final result section.

3 Code Structure and Implementation

For the purpose of this project, we have decided to implement the GA framework by ourselves, instead of using an external library. The architecture of the framework is very similar to the one developed for the first problem.

For the implementation of the GA we have opted for an object oriented approach written in C++. Additionally, we have used Boost libraries [1] for random number generation, parsers, command line parameters as well as shared pointers. Plots for the results have been created using the text results files and `gnuplot` program.

The individual is represented (encapsulated into a class) as a linear array of `unsigned int`. The arrangement of the elements has been previously explained in figure (2). We have used this particular arrangement so that elements within the same box have consecutive positions in memory. This way memory access is optimized. The access to the elements of the Sudoku board can be addressed in three different ways. First, as an absolute linear index specifying the position within the array. Second, as a pair of coordinates (box, box-offset), specifying the box number where the element is and the offset within the box. Third, as a pair of coordinates (row, column), specifying the row and column of the element within the board. Translation functions between these three coordinate systems have been provided, so that each system is used when the algorithm to be implemented is better adapted to one particular system.

Code organization is based in the following files:

- `Mutation.hpp`, implements the mutator operators. An abstract class `Mutation` has been defined, and it is derived for each mutation method. The main function within the class is the `Mutate` procedure, which given an individual, it introduces a random mutation on the genotype.
- `CrossOver.hpp`, implements the crossover operators. As for mutation, an abstract class is defined, which must be derived for each crossover method. The main function within the class is the `DoCrossover` procedure, which takes two parents and perform crossover returning two offsprings.
- `Selector.hpp`, implements the selection method for mating and surviving. As described previously four selection schemes have been implemented. The `Select` method will select a proportion of λ individuals from the population passed as argument.

- `Fitness.hpp`, implements the fitness evaluation function. An abstract class has been defined, which must be derived for each function. For this project we have created just one evaluation function, based on the number of violated constraints. The main method is the `Evaluate` function which returns the fitness value for a given individual.
- `LocalSearch.hpp` models an abstract local search algorithm (hill-climber). The abstract class is derived by the `DerivedConstraints` class which implements our local search method. The main function within the class is the `NextStep` method, which, given an individual, perform a local search returning the next individual within the search process , eventually close to the final solution and with probably a better fitness value.
- `Sudoku.hpp` class models a Sudoku Board. The Sudoku board is implemented through a linear array of `unsigned int`. Methods to access each element within the board in different coordinate systems are provided, as well as method to provide a string representation of the individual, which have been proved fundamental for debugging purposes. Each individual has the information of the position of the given values of the Sudoku Board.
- `SudokuGivens.hpp` class is just a Sudoku Board which stores the givens values of the problem. Each individual has a pointer to a unique `SudokuGivens` object to access the information of the positions of the givens values.
- `SudokuConstrained.hpp` class is a Sudoku Board where each element, instead of holding a single number, holds a set of valid numbers which fit in that position according to the constraints and the given values. It is used for the local search method.
- `Population.hpp` models the population of the GA as a linear vector of `SudokuBoard`. It provides some methods for initialization, adding individuals, removing all individuals and accessing specific individuals.
- `OptionParser.hpp` class parses the command-line arguments and returns the parameters used by the GA, as well as the `SudokuGiven` object containing the givens and the operators used for transformation and selection.
- `RandomGenerator.hpp` classes implements several kind of random generators used. *Bernoulli*, *discrete* and *uniform* random generators are implemented using Boost C++ library.
- `GA.hpp` is the orchestrator of the evolutionary process. It contains references to `Mutator`, `CrossOver`, `Evaluator`, `Selectors` and `Population` objects, as well as the parameters (mutation probability, recombination probability, maximum number of generations) required to run the simulation. It defines the methods to run the GA in an intuitive way, via `Initialize`, `NextGeneration` and `TerminationCondition` methods, which, initializes the object, runs a one-generation step in the simulation and checks for the termination condition of the algorithm respectively.
- `ResultAccumulator.hpp` object stores partial results of the population for each generation. For each step in the simulation, it keeps tracks of the best and worst individuals, the average fitness value of the population, the user-cpu time consumed, the probability distribution (histogram) of the population, as well as a copy of the best fitted individual.
- `types.hpp` Type definition used within the code.
- `utils.hpp` Utilities used through the code. Worth to mention the functions used to transform one system of coordinates into another.
- `CSVTableParser.hpp` implements a parser to read a CSV external data file with the information of the givens.

4 Final Results

After measuring the performance of each component apart, *mutation* and *crossover*, *selection*, we put all pieces together to build the final model which will be tested against several puzzles of different levels.

The experiments have a population size of 300, a maximum number of generations of 5000 and are repeated over 40 independent executions to average final results. Configuration files for the experiment are found in files `final-test-basic-yyyy-xx.cfg` and `final-test-search-yyyy-xx.cfg`, where `yyyy` is the difficulty of the puzzle (`easy`, `medium`, `hard`, `very hard`), and `xx` are the number of the puzzle. `basic` configuration files are those experiment using just mutation and crossover, while `search` configuration files use local search.

4.1 Basic Solution, mutation and crossover

As we have discussed previously, with the basic GA framework of *mutation* and *crossover* we have only been able to solve the Easy-Level with an acceptable SR, obtaining mixed values for Medium-Level puzzles, and non-acceptable solutions for Hard and Very Hard-Levels. For those non-solved puzzles, with difficulty higher than Medium, the MBF remains high, with average within the range from 10.0 to 17.0 (number of non-satisfied constraints), which means that the best individual is still far from the final solution. All numerical results can be found on table (3).

For Easy-Level puzzles, solution is found quickly, which means that intelligent operators perform well in these levels, guiding the search with the aid of the heuristic to the final solution. For the most difficult levels, the intelligent operators get stuck always on local minima, and while the diversity of the population is high, there are no chances for individuals far from this local minima to reach the final solution.

Puzzle	SR	MBF	Gen. to Solution
Easy 1	1.0	0.00	295.18
Easy 2	1.0	0.0	317.00
Easy 3	0.975	0.05	375.08
Easy 4	1.0	0.0	243.28
Easy 5	0.725	1.025	1493.41
Medium 1	0.65	3.025	2385.85
Medium 2	0.275	9.375	2295.64
Medium 3	0.325	6.9	2155.69
Medium 4	0.15	8.05	1893.00
Medium 5	0.725	1.025	3145.25
Hard 1	0.05	10.375	2323.00
Hard 2	0.0	15.75	na
Hard 3	0.025	12.7	3971.00
Hard 4	0.05	10.375	2243.00
Hard 5	0.075	17.025	1175.33
Very Hard 1	0.025	10.2	3159.00
Very Hard 2	0.0	12.55	na
Very Hard 3	0.225	9.675	2405.22
Very Hard 4	0.3	8.125	3271.83
Very Hard 5	0.0	10.975	na

Table 3: Results for the basic GA, with mutation and crossover, for different puzzle levels

4.2 Local Search

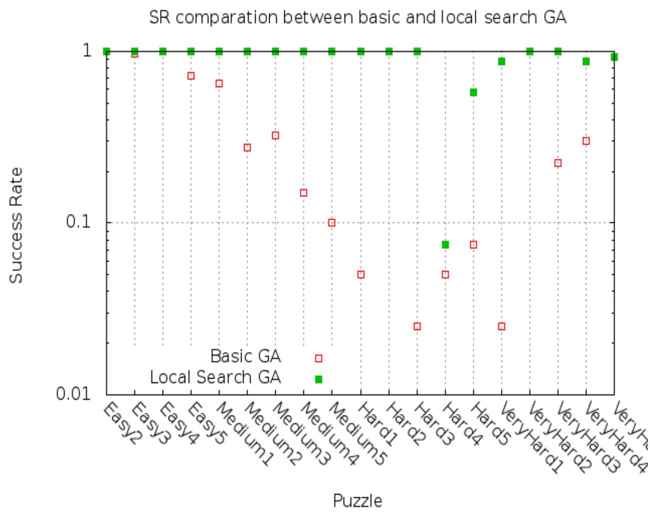
When introducing *local search* to the GA, results are improved considerably, as table (4) shows. Easy and Medium Level puzzles are all solved in the blink of a second. For Easy and Medium-level puzzles, solutions are found within the first generation, which means that the constraint propagation performed by the local search along with the intelligent mutation and crossover operator, is able to solve directly the puzzle from the very first execution.

The real improvement of the method is shown in Hard and Very Hard-Level problems, where we are now able to solve, with very acceptable SR statistic, most of the puzzles. Just two of the Hard-Level puzzles presents non-acceptable SR, and in those cases MBF is low. The number of generations needed to reach the solution is also kept very low, so, the added complexity and overhead in consumed cpu-time introduced by the local search is worth it.

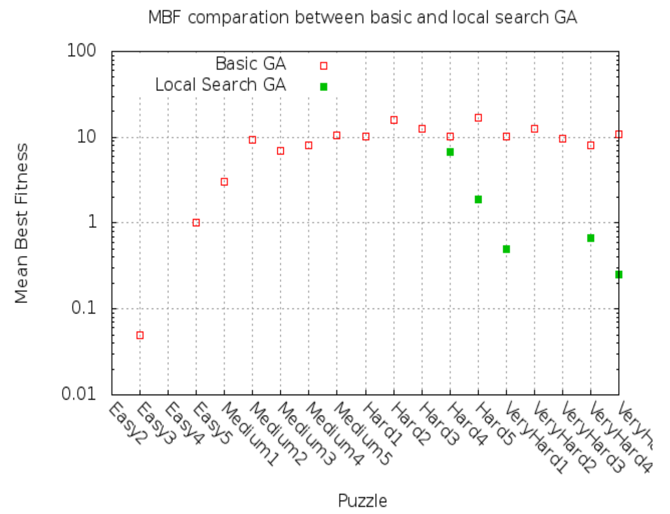
Puzzle	SR	MBF	Gen. to Solution
Easy 1	1.0	0.0	1.0
Easy 2	1.0	0.0	1.0
Easy 3	1.0	0.0	1.0
Easy 4	1.0	0.0	1.0
Easy 5	1.0	0.0	1.0
Medium 1	1.0	0.0	1.0
Medium 2	1.0	0.0	1.0
Medium 3	1.0	0.0	1.0
Medium 4	1.0	0.0	1.0
Medium 5	1.0	0.0	1.0
Hard 1	1.0	0.0	1377.90
Hard 2	1.0	0.0	413.33
Hard 3	1.0	0.0	1038.28
Hard 4	0.075	6.75	2136.33
Hard 5	0.575	1.9	1881.57
Very Hard 1	0.875	0.5	1401.34
Very Hard 2	1.0	0.0	110.20
Very Hard 3	1.0	0.0	620.90
Very Hard 4	0.875	0.675	919.60
Very Hard 5	0.925	0.25	1178.95

Table 4: Results for the GA with local search for different puzzle levels

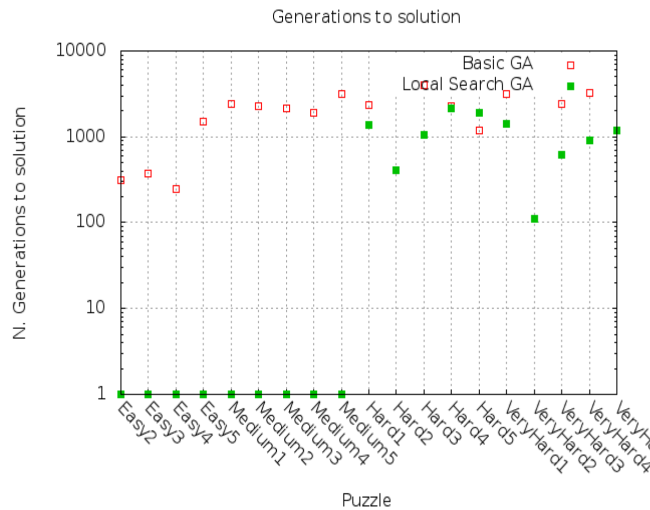
A comparation between the basic and local search method is showed in figure (11). As we can appreciate, local-search method is superior in all aspects, being able to solve successfully and in a short time even the most difficult puzzles where the basic GA failed. Thus, the relative little effort to implement a simple local search method pays off, by far, the improvements in the final results.



(a) SR for the basic GA and local search GA



(b) MBF for the basic GA and local search GA



(c) Number of generations to solution for the basic GA and local search GA

Figure 11: Comparative of performance between the basic GA (mutation+crossover) and the GA enhanced with local search (mutation + crossover + local search)

4.3 Parameter adaptation

For the Sudoku problem, we have not provided any *parameter adaptation* schema for the Sudoku Problem, however, we have thought on possible future improvements based on the results obtained with the basic configuration.

In all previous experiences we have notice that *diversity* of the population is a key measure of the success of the strategy in many of the experiments. As we have pointed out, the definition of the fitness evaluation function, which translate the *constraints* to simple numerical values, might be misleading for the GA at many times. Thus, we can have Sudoku boards with fitness evaluation as low as 2 (two constraint violations) and being very far from the solution (the number of swaps we should perform on the board to reach the final solution is very large). A population with large diversity measures can help to explore new individuals, far from the current local optimum, which eventually can lead to the final solution. On the other hand, diversity is not enough by itself to improve the results of the GA. As we saw with the fitness proportional selection method, a minimum selection pressure which retains key individuals is also compulsory to assure convergence.

In consequence, control parameter component for this GA should be focus, in our opinion, on achieving a target diversity within the population. To reach this diversity, we have identified the *mutation probability* and the *shape-factor* controlling the selection pressure as the key parameters to adjust. *Cross-over probability*, as we saw in previous section, has a much more complex relationship, and thus, its impact on the final diversity and quality of solution is much more complex to model in control parameter algorithm.

Our intuition, based on previous results, is set to think that the *control parameter* implementation could eventually improve the solution for the basic GA set-up (with just *mutation* and *crossover*), where solutions are found just for Easy-Level puzzles and some Medium-Level. However, in case we use *local search*, a *control parameter* would not add significant improvement in the performance of the algorithm since, as we have tested, this GA add-in is, by itself, able to solve most of the Sudoku puzzles used in the experiment with little effort in the implementation, in terms of code-complexity and fine tuning of parameters (our local search method has not any parameter associated with the algorithm).

4.4 Conclusion

We have successfully solved a wide variety of Sudoku puzzles of different levels, from Easy to Very Hard problem, according to the classification on `www.dailysudoku.com` website. While the classic GA with just *mutation* and *crossover* operator is enough to solve the Easy-Level puzzles, more advanced techniques must be employed to solve the most difficult problems.

On the *mutation* operator, we have seen that implementing several method is very useful, since a *combined* operator can show synergies and out performance each of the individual operators taken apart. We have also shown how operators with specific knowledge of the problem, instead of just random method, can improve considerably the results of the algorithm with little effort on the implementation.

As for the *crossover* operator, we have also used the *combined* operator approach, using one pure random method, *swap crossover*, and other more intelligent operator which make use of superior knowledge of the problem to be solved to create off-springs with optimized fitness values. The introduction of this method increased the SR results for the Medium-Level puzzles, which mutation operator by itself was unable to solve. However, the impact of the probability of crossover on the final results is not so clear as in the *mutation probability*, and a more detailed analysis at the *crossover probability* as well as the individual probabilities of each method in the combined operator has been necessary.

When fine tuning the above operators, looking for optimal configuration, we have been forced to do lots of experiments varying each parameter independently in order to conclude some more favorable

parameter-set. This point highlight the importance of using a *control parameter* module in our GA which could modify this parameters on the fly until reaching an optimal configuration. Unfortunately, we have not had enough time to implement this important module, which is left for future improvements.

A simple *fitness evaluation function* which counts the number of violated constraints has been implemented in order to translate the CSP to one easily tackled by GA. Although it seems to be the standard method used in most articles devoted to solving Sudoku problems with GA ([4], [5]), in our opinion, it shows some lack of efficiency in many situations, leading to local-optimal solutions. More specifically, we have found many cases where Sudoku boards with fitness values as low as 2 are far from the optimal solution. If we think of a *distance function* between a given Sudoku board and its final solution in terms of the number of swaps of elements it would be necessary to perform in order to reach the solution, it would be a desirable property that the *fitness evaluation function* would be consistent with this *distance function*. Unfortunately, such a *distance function* is not easy to derive since, in the general case, we would need to know the final solution in order to calculate the distance. Our intuition is that, diving deeper in the mathematical theory of Sudokus, we might come up with some kind of heuristic function as an approximation of this desirable *distance function*. This heuristic would, in our opinion, lead to the definition of a more consistent fitness evaluation function, thus, allowing the GA to reach the final solution much faster.

All in all, with the basic GA we have successfully solved the Easy-Level puzzles, but we have failed to solve the most complex one. Being GA not the best method to solve CSP, we have needed to employ the *local search* add-in in order to tackle the most challenging puzzles. *Local search* method has been implemented with little effort, analyzing in three basic steps the possible set of values that each cell in the grid can take. Then, the *local search* is done by moving each individual within the population to the nearest element which satisfies the constraints derived for each cell. With this simple algorithm, we have successfully solved, with good SR puzzles from all categories without the effort of implementing a complete and exact algorithm used in CSP.

Summarizing, even if more superior methods exist to solve CSP, we have successfully solved the Sudoku problem with GA. The use of *hybridization techniques*, instead of a pure random GA, such as intelligent *mutation* and *crossover* operators, as well as a *local search* module, has made possible the solution of the problem with little effort.

4.4.1 Improvements

There are still some ideas that we think should be studied more in detail. First, the use of non-traditional *fitness evaluation function* using heuristic derived from theoretical results could improve significantly the performance, although their implementation might require a superior understanding of the mathematics of Sudoku.

More advanced *crossover* operator should be investigated. In our opinion, the two implemented *crossover* methods do not provide significant improvement to the transformation performed by *mutation*. Some other *hybridization* technique should be implemented. On the other hand, we have not studied the impact of varying the number of *crossover* points for the *swap* operator, which eventually might have some impact on the final performance.

Finally, a *control parameter* module would be highly desirable. Even if we think that it would not add any significant improvement to the final GA using *local search*, it might be very useful for the basic GA set-up with just *mutation* and *crossover* operators.

5 Appendix

A Program Compilation

Program building process has be done using the standard make gnu tool. To compile and link the program just invoke the `make` command on the root directory. The binaries will be found either under the `bin/debug` or `bin/release` directories, depending on the `dbg` flag specified at `config.mk` file. To link the program and generate the binary, we need the Boost 1.46 Libraries properly installed in our system. The `libraries.mk` file contains the list of libraries that we need for the linking process.

B Program Execution

The program `sudoku.bin`. is invoked from the command line, passing all required parameters needed to the execution of the algorithm. For a comprehensive list of parameters, we can invoke:

```
bin/release/sudoku.bin --help
Allowed options:
  --conf_file arg                        Path to the configuration file. If
                                         omitted, arguments are read from
                                         command line
  --givens_file arg                      Path to the givens file, comma separated values
  --mutation arg                         Mutator operator: [combined|swap|swap3|
                                         intelligent]
  --mutationProb arg (=0.30)
                                         Mutation Probability
  --mutationSwapProb arg (=0)            Mutation Probability for swap operator
  --mutationSwapProbBox arg (=0)         Mutation Probability for an element
                                         within a box
  --mutationSwap3Prob arg (=0)           Mutation Probability for swap 3
                                         operator
  --mutationSwap3ProbBox arg (=0)        Mutation Probability for an element
                                         within a box
  --mutationIntelligentProb arg (=0)     Mutation Probability for intelligent
                                         operator
  --crossover arg                        CrossOver operator: [boxXover|optimazed
                                         rowcol|combined]
  --crossoverProb arg (=0)               CrossOver probability
  --crossoverSwapProb arg (=0)           CrossOver probability for swap operator
  --crossoverSwapProbBox arg (=0)        CrossOver probability for a box
  --crossoverOptimizedProb arg (=0)      CrossOver probability for optimized
                                         operator
  --localSearch arg                      Use Local Search: [derivedconstraints]
  --selection arg                        Selection operator:[ranking|fitnessprop
                                         ortional|tournamentElitism|rankingElitism]
  --probabilityShape arg (=0.30)
                                         Probability Shape factor for selection
                                         method
  --tournamentSize arg (=2)              Tournament Size
  --alphaElitism arg (=0.10)
                                         Elitism for the selection, the best
                                         alpha% will be chosen
  --tournamentProb arg (=0.75)           Probability for the Stochastic
```

```
                                       Tournament Selection
  --evaluation arg                     Evaluation operator:[numOfViolations]
  --mu arg (=30)                       Number of individuals in the population
  --lambda arg (=200)                  Number of offsprings created at each
                                       generation
  --generations arg (=1200)            Number of generations for the
                                       termination condition
  --executions arg (=10)               Number of independent executions
  --outputFile arg (=results.txt)      Filename for output data
  --test                               Run mutation and crossover tests for
                                       debuggin purposes
  --help                               print out help message
```

In order to easily define simulation, we can define all the above parameters in a configuration file and pass this file as the argument `--conf-file arg`. All experiments run to create this report have been saved in a configuration file with suffix `.cfg`, saved under the `results` folder. Refer these files for further examples.

C Sudoku Puzzles

All Sudoku puzzles used for the experiment in this problem have been obtained from `www.dailysudoku.com`. The Sudoku puzzles can be found under the directory `puzzles`. The file name shows their difficulty level.

The format for the file containing the Sudoku puzzles is a common `CSV` file, using `;` character as separator. No blank line should exist after the last row of the sudoku puzzles when reading the puzzle from a file.

D Acronyms

CSP Constraint Satisfaction Problems 3

GA Genetic Algorithm 3

EC Evolutionary Computing 5

MBF Mean Best Fitness 9

MNFES Mean Number of Fitness Evaluation To Solution 9

SR Success Rate 9

References

[1] Robert Demming and Daniel J. Duffy. *Introduction to the Boost C++ Libraries, Volume I - Foundations*. Datasim Education BV, 2010.

[2] A. E. Eiben and J. E. Smith. *Introduction to evolutionary computation*. Natural computing series. Springer-Verlag, 2003.

[3] Donald E Knuth. Dancing links. *Millenial Perspectives in Computer Science*, 18:4, Sep 2009. Comments: Abstract added by Greg Kuperberg.

[4] Timo Mantere and Janne Koljonen. Solving, rating and generating sudoku puzzles with ga. In *IEEE Congress on Evolutionary Computation*, pages 1382–1389. IEEE, 2007.

[5] Yuji Sato and Hazuki Inoue. Solving sudoku with genetic operations that preserve building blocks.

[6] Julienne Walker. The art of hasing. `http://eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx`.